CPSC 3720: Lecture : Introduction to Design & UML Class Diagrams

Eileen T. Kraemer and Murali Sitaraman, Clemson

E-mails: etkraem@clemson.edu and murali@Clemson.edu

Fall 2018 Version

Some of the lecture slides have benefitted from the findings of grants from the US National Science Foundation.

Software Lifecycle Phases

Requirements Analysis

Design & Specification

Implementation

Quality Assurance

Maintenance

Discussion: Purpose of design

Design & Specification Intro

DESIGN &
Specification

- Objectives: HOW?
- Inputs
- Outputs
- Approaches
- Verification

Implementation

Quality
Ass

Maintenance

Design Intro

- Objectives: HOW should this system be built?
  - Key points: Modularization, reuse
- Inputs: Requirements Definition Document (RDD)
- Outputs: Design Documents
  - Design diagrams and specifications of modules
- Example Approaches:
  - Structured design
  - Object-oriented design
  - Component-based design

Design Principles

- Coupling
  - How inter-related are the modules in a system?
- Cohesion
  - How “singled-minded” is each module?

Coupling

- Coupling is communication
- Design goal: Minimize coupling, so there is less need for communication among modules
- All coupling cannot be avoided.
- Design goal: Where coupling is unavoidable:
  - make it a "desirable form" of coupling
  - Make the communication that is needed precise

Undesirable Forms of Coupling

- They are “non-modular”; Make it difficult to reason about software
- Global (or Common) coupling
  - E.g., Use global variables to share information among modules
- Content coupling
  - E.g., Implementation inheritance, whereby changes to the content of a class implementation affects that of another
- Control coupling
  - An external flag controls flow in a module

Desirable Forms of Coupling

- Parametric coupling
  - E.g., use of parameters to communicate information among modules
- Coupling strictly through interfaces
  - Reuse of components strictly based on their interface specifications
  - Specification inheritance, whereby existing specification is extended

Cohesion

- Cohesion is a module being "coherent"; i.e., making sense
- Design goal: Maximize cohesion in each module
- Design goal: Make the cohesion to be a "desirable form" of cohesion

Undesirable Forms of Cohesion

- Grouping of "unrelated" elements
- Coincidental cohesion
  - E.g., elements of a module come together by accident
- Logical cohesion
  - Sounds better than it actually is.
  - E.g., All outputs are grouped in a module
- Temporal cohesion
  - Elements that happen in close proximity in time are grouped together
  - E.g., "A start up" module

Desirable Forms of Cohesion

- Functional cohesion
  - Elements in a module perform related functionality
  - E.g., Interface specification and implementations that capture a well-designed "abstract data type" (e.g., stacks, queues, lists, or maps)

DESIGN WITH UML CLASS DIAGRAMS

Topics:

- Need for abstraction in communicating and reasoning about designs
- Brief introduction to UML notations
- New modeling abstractions provided by UML

Motivation: Reasoning about a design

- **Goal:** Be able to "reason about" a design
  - i.e., understand designer's intent
  - Critique/improve the design
- **Claim:** Source code not best medium for communication and comprehension
  - Lots of redundancy and detail irrelevant for some program-understanding tasks
  - Especially poor at depicting *relationships* among classes in OO programs
  - To understand an OO design, one must be able to visualize these relationships
- **Solution:** Use abstract, visual representations

Unified Modeling Language (UML)

- Collection of notations representing software designs from three points of view:
  - ***Class model*** describes the static structure of objects and relationships in a system
    - Comprises *object* and *class diagrams*
    - Provides new and very useful abstractions for reasoning
  - ***State model*** describes the dynamic aspects of objects and the nature of control in a system
  - ***Interaction model*** describes how objects in a system cooperate to achieve broader results
- Generally need all three models to describe a system
- No single model says everything

Unified Modeling Language (UML)

- Collection of notations representing software designs from three points of view:
  - ***Class model*** describes the static structure of objects and relationships in a system
    - Comprises *object* and *class diagrams*
    - Provides new and very useful abstractions for reasoning
  - ***State model*** describes the dynamic aspects of objects and the nature of control in a system
  - ***Interaction model*** describes how objects in a system cooperate to achieve broader results
- Generally need all three models to describe a system
- No single model says everything

UML class diagram notation

- Boxes denote classes
- Each box comprises:
  - Class name (e.g., Employee)
  - List of data *attributes* (e.g., first_name, last_name, etc).
  - List of *operations* (e.g., print)

Employee
first_name : string last_name : string hire_date : Date department : short
print(ostream&):void

City
cityName : string population : unsigned

Attributes

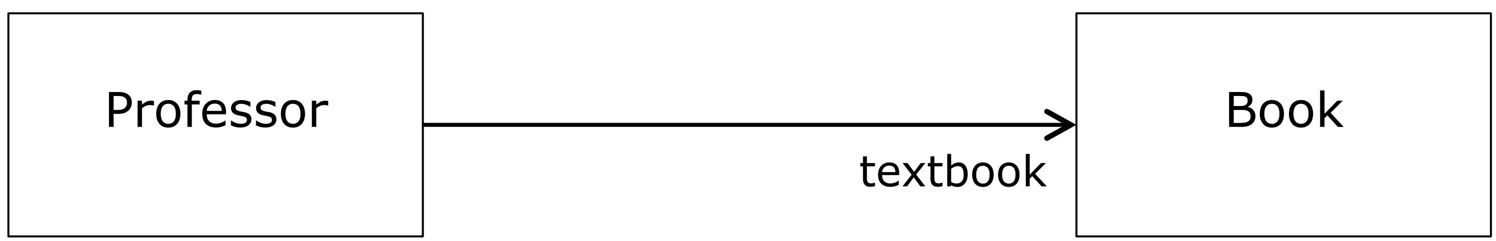
- Full form:

`visibility name: type multiplicity = default {property-string}`

- **visibility**: + or – or # or ~
  - to indicate public or private or protected or package
- **name**: name of field in class or name of association
- **type**: what kind of object can be placed here
- **multiplicity:** see later slides
- **default**: the value of a newly created object if the attribute value is not specified when created
- **{property-string}:** can indicate additional properties using certain keywords (e.g. ReadOnly)

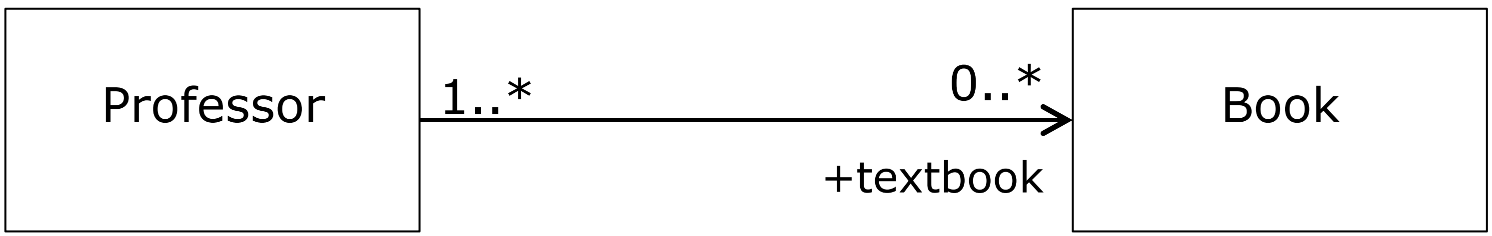
Associations

- Another way to notate a property
- Similar to an attribute, but less like a field in an object and more like a relationship to other classes or objects
- A solid line between two classes, directed from source class to target class



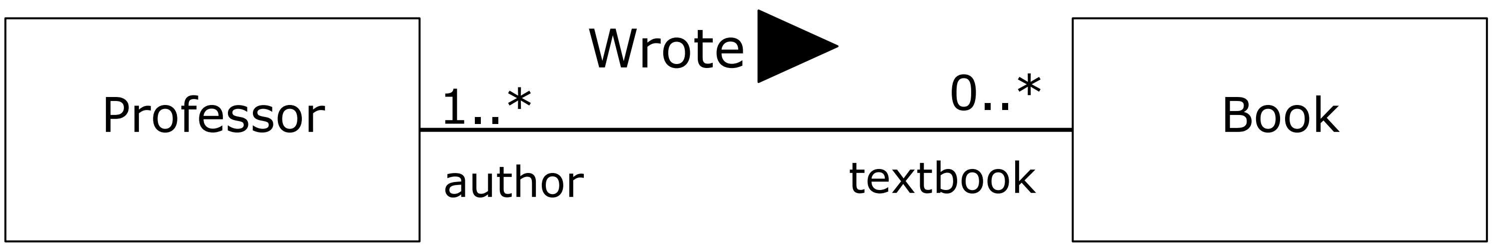
Associations

- Another way to notate a property
- Similar to an attribute, but less like a field in an object and more like a relationship to other classes or objects
- A solid line between two classes, directed from source class to target class



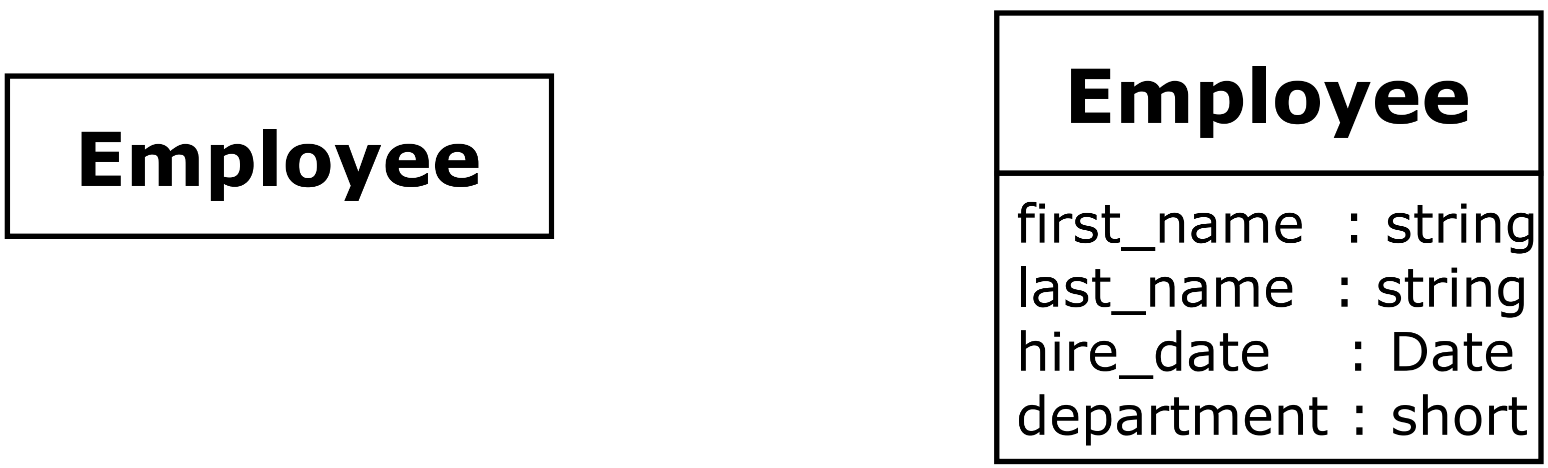
Associations

- You might also see this notation



Abstraction in class diagrams

- Class diagrams often elide details:
  - Method associated with an operation
  - Attributes and operations may be elided in the diagram to improve readability
    - Even if they exist in the code



Example: Travel-planning system

City
cityName : string population : unsigned timeZone : zone airportName : string airportCode : code

Consider class City

Question: Can you find a flaw in this design?

Example: Travel-planning system

Consider class City

Question: Can you find a flaw in this design?

City
cityName : string population : unsigned timeZone : zone airportName : string airportCode : code

Answer: These attributes "hiding" an object (i.e., an airport) that is meaningful in this domain

Question

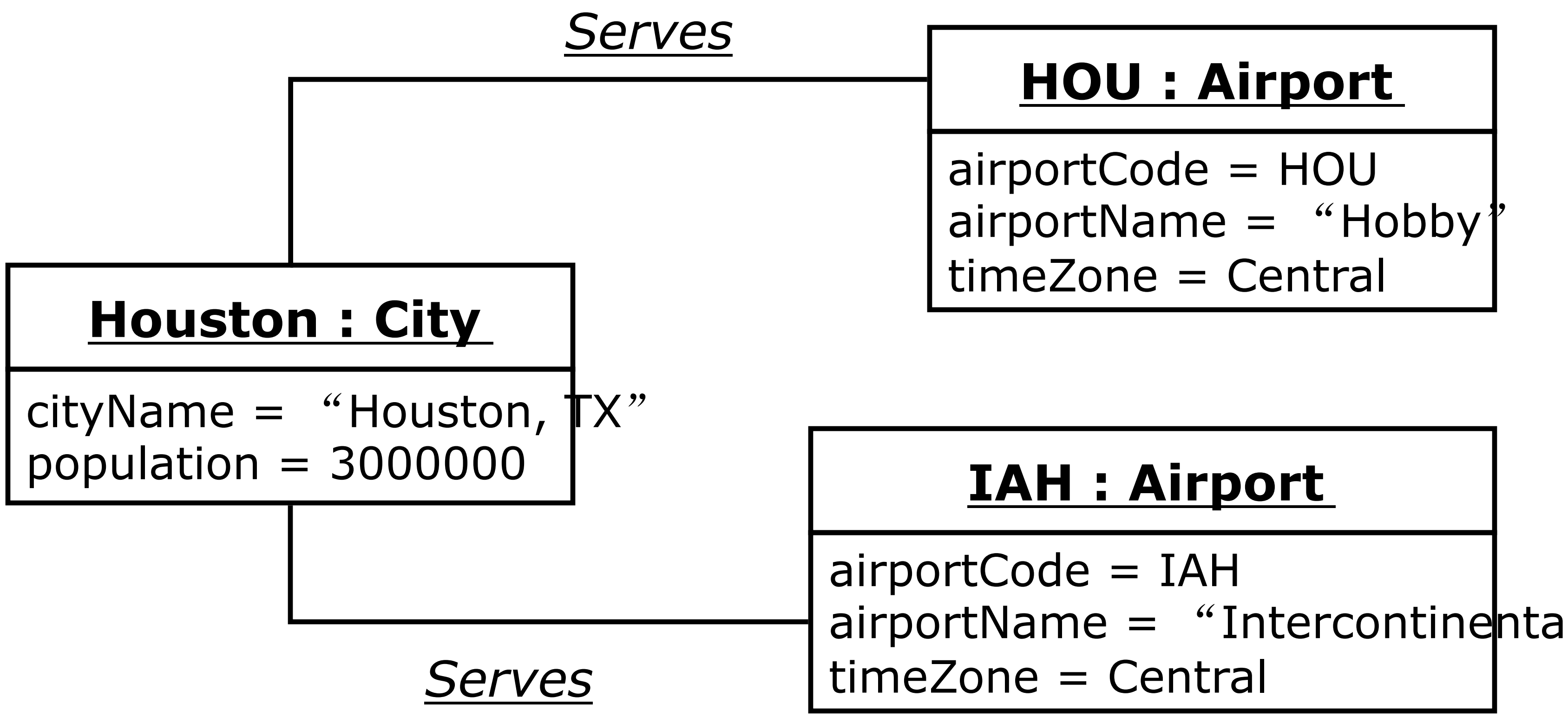
- Why might it be bad to encode one class as a collection of attribute values within another?

Question

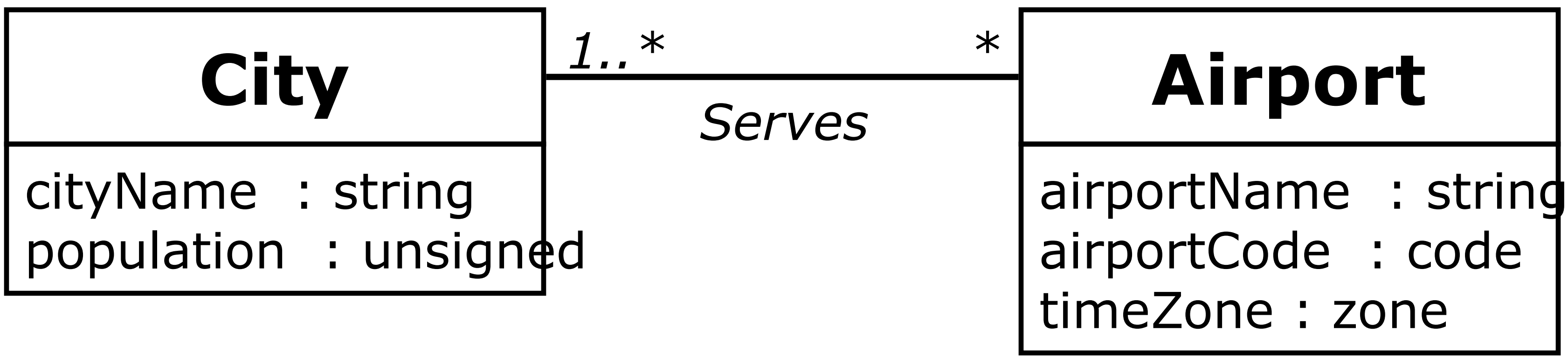
- Why might it be bad to encode one class as a collection of attribute values within another?

- Answers:
  - Potential for redundancy/inconsistency due to duplication
    - some airports serve multiple cities
    - some cities served by no airports
    - some cities served by multiple airports
  - Operations over Airport objects may not need to know details associated with cities, such as population

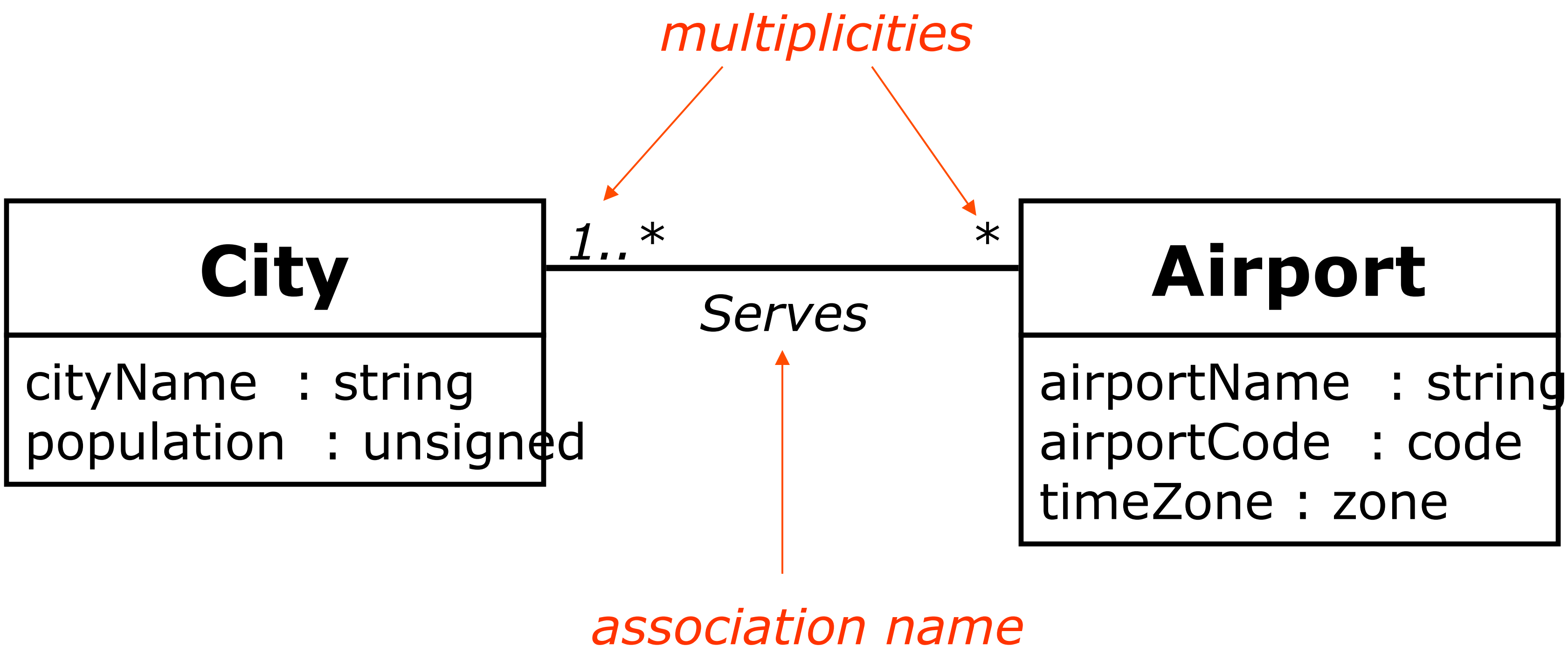
Solution: Use of Associations



Example association



Example association



New abstraction: Bidirectionality

- Links may be *navigated* in either direction!
- Benefits:
  - During early design, it is often difficult to predict the navigation directions that will be needed
    - Especially true for many-to-many associations
    - Better to model connections as bidirectional associations and later *refine* these associations into more implementation-level structures (e.g., pointers, vectors of pointers, maps, etc)
  - Often several ways to implement an association and the details are not salient to the "essence" of the design

Example: Refinements of Serves association


```
class City {
 ...
 protected:
  string cityName;
  unsigned population;
  vector<Airport*> serves;
};

class Airport {
  ...
 protected:
  string airportName;
  CODE airportCode;
  ZONE timeZone;
  vector<City*> serves;
};
```

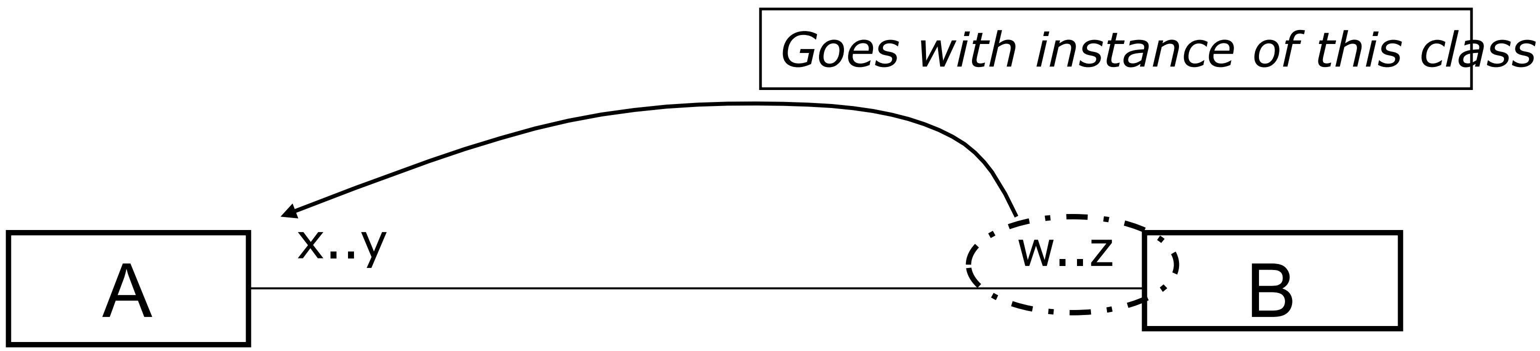
```
class City {
 ...
 protected:
  string cityName;
  unsigned population;
};

class Airport {
  ...
 protected:
  string airportName;
  CODE airportCode;
  ZONE timeZone;
};
multimap<City*, Airport*> cityServes;
multimap<Airport*, City*> airportServes;
```

Design tip

- You should get comfortable with the various methods for refining a UML association
  - be able to easily switch back and forth between what is said in the diagram and what is allowable in the code
  - start to “think” using links/associations rather than pointers and references
- This is good training in abstraction

New abstraction: Multiplicity constraints



There are
x to *y* A’ s
for each B

There are *w* to *z* B’ s
for each A

x ⇒ 0, 1, or specific number

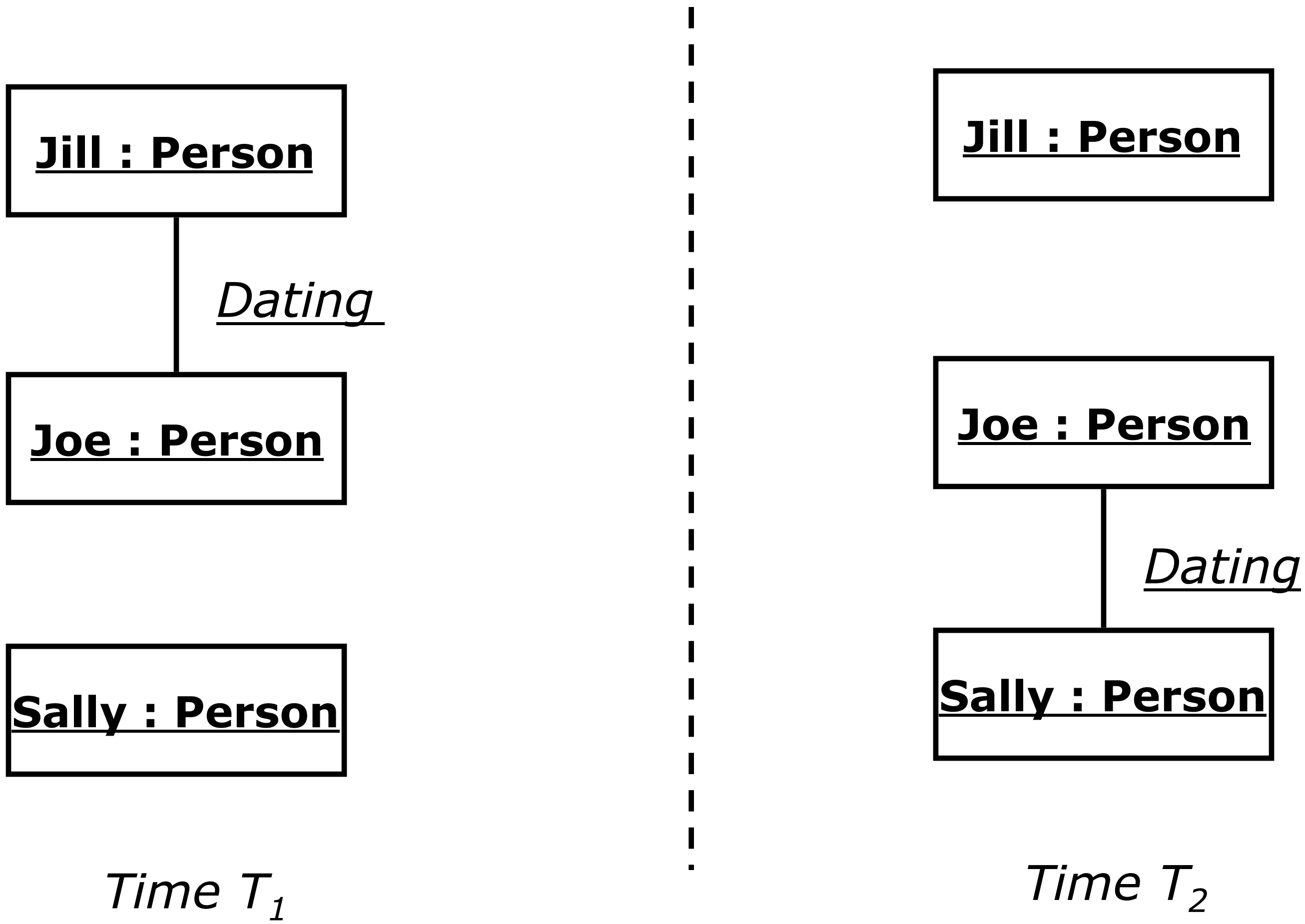
y ⇒ 1, *, or specifc number
* = “any number”

Also: A specific list is acceptable. E.g., 2, 4, 6

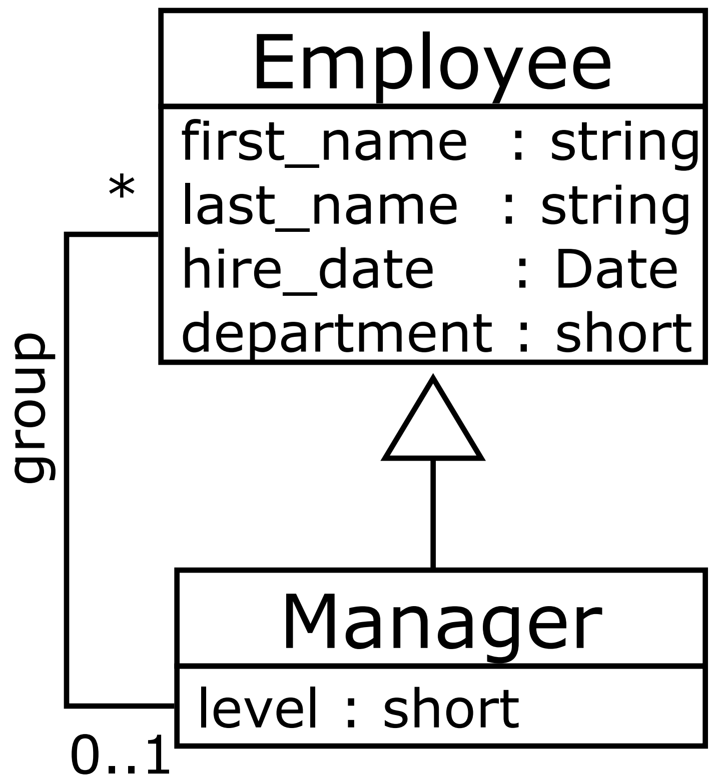
Common Multiplicity Idioms

0..1	“Optional”
1..*	“At least one”
0..*	“Any number”

Object diagrams are “snapshots”



New abstraction: Generalization



- A.k.a. the "is-a" relation
- Relates class to one that is "more general"
  - Open arrow points to base class (i.e., the generalization)
  - Derived class inherits all attributes/operations of base class
- Relation is anti-symmetric and transitive

Summary

- UML provides notations for modeling objects and classes of a system
- Diagrams denote models:
  - more abstract than implementations
    - E.g., links vs. pointers
    - E.g., associations describe collections of links
  - useful for explanation/documentation
- As we shall see, class models also useful prior to implementation

Exercise

- Customer/Order/Product example ...